(1) kd

ORIGINAL



OFFICE OF UNIVERSITY COUNSEL
James J. Mingle (Bar Roll No. 508993)
Nelson E. Roth (Bar Roll No. 102486)
300 CCC Building, Garden Avenue
Cornell University
Ithaca, New York 14853
Tel: 607-255-5124

SIDLEY AUSTIN BROWN & WOOD
Edward G. Poplawski
Denise L. McKenzie
555 West Fifth Street, 40th Floor
Los Angeles, California 90013
Tel: 213-896-6000

SIDLEY AUSTIN BROWN & WOOD
James D. Arden (JA 8779)
John J. Lavelle (JL 1455)
875 Third Avenue
New York, New York 10022
Tel: 212-906-2000

Attorneys for Plaintiffs
Cornell University and
Cornell Research Foundation, Inc.



**IN THE UNITED STATES DISTRICT COURT**
**FOR THE NORTHERN DISTRICT OF NEW YORK**

01 -CV- 1974

| | |
|---|---|
| CORNELL UNIVERSITY, a non-profit New York corporation, and CORNELL RESEARCH FOUNDATION, INC., a non-profit New York corporation, | Case No. _____ |
| Plaintiffs, | **COMPLAINT FOR INFRINGEMENT OF UNITED STATES PATENT NO. 4,807,115** |
| v. | **DEMAND FOR JURY TRIAL** |
| HEWLETT-PACKARD COMPANY, a Delaware corporation, | |
| Defendant. | |

Plaintiffs, Cornell University and Cornell Research Foundation, Inc., bring this civil action against Hewlett-Packard Company and hereby aver and complain as follows:

**Averments Common To All Claims For Relief**

1. This action is for injunctive relief and damages and arises under the United States patent laws (35 U.S.C. §§ 271, *et seq.*). This Court has subject matter jurisdiction under 28 U.S.C. §§ 1331 and 1338.

2. Venue is proper in this judicial district under 28 U.S.C.§ 1400(b).

3. Plaintiff Cornell University ("Cornell") is an educational institution established in 1865 and is incorporated under the laws of the State of New York, with its main campus and place of business in Ithaca, Tompkins County, New York. Cornell's charter is contained at Article 115 of the New York Education Law.

4. Plaintiff Cornell Research Foundation, Inc. ("Cornell Research") is a New York non-profit corporation, having its principal place of business at 20 Thornwood Drive, Suite 105, Ithaca, New York 14850. Cornell Research is a wholly owned subsidiary of Cornell University whose mission is to manage the intellectual property on behalf of Cornell

University, including obtaining patent, trademark or copyright protection where appropriate and licensing intellectual property for commercial development and use.

5. On information and belief, Defendant Hewlett-Packard Company ("Hewlett-Packard") is a Delaware corporation, having its principal place of business at 3000 Hanover Street Palo Alto, California 94304. Hewlett-Packard regularly conducts and transacts business in New York, throughout the United States and within this judicial district, and as set forth in paragraphs 16-21 below, has committed, and continues to commit, tortious acts of patent infringement within and outside of New York and within this judicial district. Hewlett-Packard further has engaged, and continues to engage, in continuous, permanent, and substantial activity in New York. Hewlett-Packard is licensed to do business in New York and has one or more places of business in this judicial district.

6. On February 21, 1989, United States Patent No. 4,807,115 ("the '115 patent"), for an instruction issuing system and method for processors with multiple functional units, was duly and legally issued to Cornell Research, as assignee of the name inventor, Hwa C. Torng who resides in California. A true and correct copy of the '115 patent is attached hereto as Exhibit 1 and is incorporated by reference.

7. Since the date of the issuance of the '115 patent, Cornell Research has been and still is the owner of all right, title and interest in and to the '115 patent by assignment, including the right to sue and recover any and all damages for infringement and obtain injunctive relief.

8. Hewlett-Packard has been and still is making, offering for sale, selling, using and otherwise making available products, systems and apparatus that infringe the '115 patent, all without the authorization of Cornell and Cornell Research or either of them. One such product, system or apparatus is Hewlett-Packard's CPU known as the PA-8000. A true and correct copy of a 1995 IEEE paper authored by Doug Hunt, entitled "Advanced Performance Features of the 64-bit PA-8000," is contained in Exhibit 2 and is incorporated by reference. According to Hewlett-Packard, Exhibit 2 contains an accurate and complete description of the PA-8000. A true and correct copy of a 1996 IEEE technical paper digest authored by Neela Bhakta Gaddis, *et al* and relating to the PA-8000 is contained in Exhibit 3 and is incorporated by reference. According to Hewlett-Packard, Exhibit 3 contains an accurate and complete description of the PA-8000.

9. Hewlett-Packard has been and still is performing, implementing and carrying out processes, methods or systems that

infringe the '115 patent, all without the authorization of Cornell and Cornell Research or either of them.

10. Hewlett-Packard has also offered for sale, sold, made, used and otherwise made available, and continues to offer for sale, sell, make, use and otherwise make available, products, systems and apparatus, the operation of which necessarily infringes the '115 patent, all without the authorization of Cornell and Cornell Research or either of them.

11. Hewlett-Packard has been and still is offering for sale, selling, making, using and otherwise making available products, systems and apparatus for use in carrying out, performing or practicing a process or method of the '115 patent, and which constitute a material part of the invention of the '115 patent, knowing the same to be especially made or specially adapted for use in an infringement of the '115 patent, and not a staple article or commodity of commerce suitable for substantial non-infringing use, all without the authorization of Cornell and Cornell Research or either of them.

12. On information and belief Hewlett-Packard has been and still is actively inducing one or more third parties to infringe the '115 patent, all without the authorization of Cornell and Cornell Research or either of them.

13. Hewlett-Packard has both actual and constructive notice of the '115 patent and of its infringement of the '115 patent.

14. On information and belief the acts of Hewlett-Packard set forth above have been willful, wanton and deliberate.

15. The harm to Cornell and Cornell Research resulting from the above acts of Hewlett-Packard as set forth above is irreparable, continuing, not fully compensable in money damages and will continue unless Hewlett-Packard is enjoined by this Court.

**First Claim For Relief**

**(Direct Infringement Of The '115 Patent)**

16. Cornell and Cornell Research incorporate by reference in this claim for relief the averments contained in paragraphs 1-15 above.

17. The acts of Hewlett-Packard as described above constitute direct infringement of the '115 patent in violation of Section 271(a) of Title 35, United States Code.

**Second Claim For Relief**

**(Inducing Infringement Of The '115 Patent)**

18. Cornell and Cornell Research incorporate by reference in this claim for relief the averments contained in Paragraphs 1-15 above.

19. The acts of Hewlett-Packard as described above constitute inducing infringement of the '115 patent in violation of Section 271(b) of Title 35, United States Code.

**Third Claim For Relief**

**(Contributory Infringement Of The '115 Patent)**

20. Cornell and Cornell Research incorporates by reference in this claim for relief the averments contained in Paragraphs 1-15 above.

21. The acts of Hewlett-Packard described above constitute contributory infringement of the '115 Patent in violation of Section 271(c) of Title 35, United States Code.

**Prayer For Relief**

**WHEREFORE**, Cornell and Cornell Research pray for judgment as follows:

1. A preliminary and permanent injunction against continued infringements of the '115 patent by Hewlett-Packard

and any and all persons acting in privity or concert with it or otherwise controlled by it.

2. An award to Cornell and Cornell Research of their damages and injuries caused by Hewlett-Packard's acts.

3. An adjudication that this is an "exceptional case" and, accordingly,

i. That the damages awarded Cornell and Cornell Research be increased three (3) times pursuant to 35 U.S.C. § 284, and

ii. That Cornell and Cornell Research be awarded their reasonable attorneys' fees pursuant to 35 U.S.C. § 284.

4. An award of prejudgment and postjudgment interest on any and all damages awarded to Cornell and Cornell Research.

5. Any other relief that this Court may deem appropriate or that is otherwise proper.

Respectfully Submitted,

DATED: December ___, 2001 OFFICE OF UNIVERSITY COUNSEL

By:_____

James J. Mingle (BRN 508993)
Nelson E. Roth (BRN 102486)
300 CCC Building, Garden Avenue
Cornell University
Ithaca, New York 14853
Tel: 607-255-5124

DATED: December 20, 2001 SIDLEY AUSTIN BROWN & WOOD

By Edward G Poplawski

SIDLEY AUSTIN BROWN & WOOD
Edward G. Poplawski
Denise L. McKenzie
555 West Fifth Street, 40th Fl
Los Angeles, California 90013
Tel: 213-896-6000
- and -
James D. Arden (JA 8779)
John J. Lavelle (JL 1455)
875 Third Avenue
New York, New York
Tel: 212-906-2000

JURY TRIAL DEMAND

Plaintiffs, CORNELL UNIVERSITY and CORNELL RESEARCH FOUNDATION, INC., hereby demand trial by jury.

Respectfully Submitted,

DATED: December 21, 2001 OFFICE OF UNIVERSITY COUNSEL

By: _____

James J. Mingle (BRN 508993)
Nelson E. Roth (BRN 102486)
300 CCC Building, Garden Avenue
Cornell University
Ithaca, New York 14853
Tel: 607-255-5124

DATED: December 20, 2001 SIDLEY AUSTIN BROWN & WOOD

By _____

SIDLEY AUSTIN BROWN & WOOD
Edward G. Poplawski
Denise L. McKenzie
555 West Fifth Street, 40th Fl
Los Angeles, California 90013
Tel: 213-896-6000
- and -
James D. Arden (JA 8779)
John J. Lavelle (JL 1455)
875 Third Avenue
New York, New York
Tel: 212-906-2000

# EXHIBIT “1"

# United States Patent [19]

**Torng**

[11] **Patent Number: 4,807,115**

[45] **Date of Patent: Feb. 21, 1989**


[54] **INSTRUCTION ISSUING MECHANISM FOR PROCESSORS WITH MULTIPLE FUNCTIONAL UNITS**


[75] Inventor: **Hwa C. Torng**, Ithaca, N.Y.

[73] Assignee: **Cornell Research Foundation, Inc.**, Ithica, N.Y.



[21] Appl. No.: **112,020**

[22] Filed: **Oct. 14, 1987**

**Related U.S. Application Data**

[63] Continuation of Ser. No. 539,854, Oct. 7, 1983, abandoned.

[51] **Int. Cl.**[4] ..... **G06F 13/00**

[52] **U.S. Cl.** ..... **364/200**

[58] **Field of Search** ... 364/200 MS File, 900 MS File



[56] **References Cited**

U.S. PATENT DOCUMENTS

| | | | |
|---|---|---|---|
| 3,297,999 | 1/1967 | Shimabukuro | 364/200 |
| 3,346,851 | 10/1967 | Thornton | 364/200 |
| 3,462,744 | 8/1969 | Tomasulo et al. | 364/200 |
| 3,718,912 | 2/1973 | Hasbrouck et al. | 364/200 |
| 3,962,706 | 6/1976 | Dennis | 364/900 |
| 4,050,058 | 9/1977 | Garlic | 364/200 |
| 4,128,880 | 12/1978 | Cray | 364/200 |
| 4,179,734 | 12/1979 | O'Leary | 364/200 |
| 4,197,589 | 4/1980 | Cornish | 364/900 |
| 4,466,061 | 8/1984 | De Santis | 364/200 |

OTHER PUBLICATIONS

R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM Journal, Jan. 1967.

R. M. Keller, "Look-Ahead Processors", Computing Surveys, vol. 7, No. 4, Dec. 1975.

J. W. Bowra and H. C. Torng, "The Modeling and Design of Multiple Function-Unit Processors", IEEE Transactions on Computers, vol. C-25, No. 3, Mar. 1976.

Siewiorek, D. P. "Computer Structures: Principles and Examples", 1982, pp. 278, 288–292.

H. C. Torng et al., "An Instruction Issuing Approach to Enchancing Performance in Multiple Functional Units Processors", *IEEE Transactions on Computers*, vol. C-35, No. 9, Sep. 86.

J. E. Thornton, "Parallel Operation in the Control Data", *A FIES Proceedings*, vol. 26, pt. 2, 1964, pp. 489–496.

G. Bell et al., "The Cray-1 Computer System", Comm. of the ACM, vol. 21, No. 1, Jan. 1978.

V. P. Srinii and J. F. Asenjo, "Analysis of Cray-1S Architecture", ACM, 1983.



*Primary Examiner*—Raulfe B. Zache
*Assistant Examiner*—Florin Munteanu
*Attorney, Agent, or Firm*—Sughrue, Mion, Zinn, Macpeak & Seas



[57] **ABSTRACT**

An instruction issuing mechanism for boosting throughput of processors with multiple functional units. A Dispatch Stack (DS) and a Precedence Count Memory (PCM) are employed which allow multiple instructions to be issued per machine cycle. Additionally, instructions do no have to be issued according to their order in the instruction stream, so that non-sequential instruction issuance occurs. In this system, multiple instruction issuance and non-sequential instruction issuance policies enhance the throughput of processors with multiple functional units.


**19 Claims, 1 Drawing Sheet**

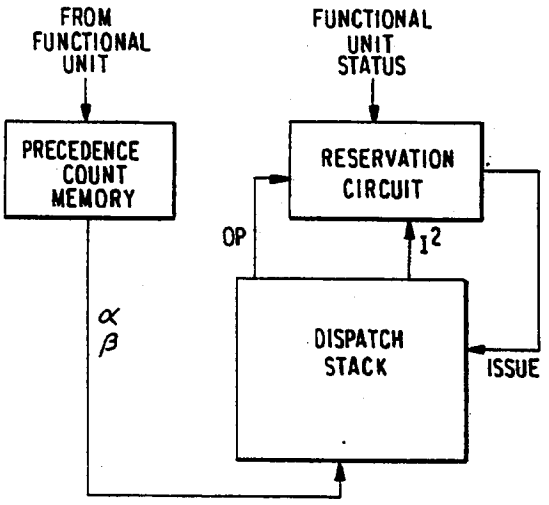


FIG.1A
PRIOR ART

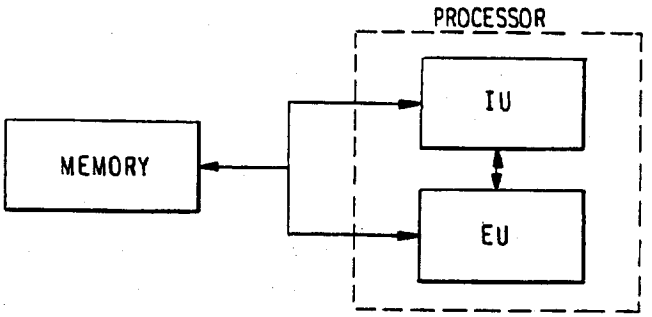
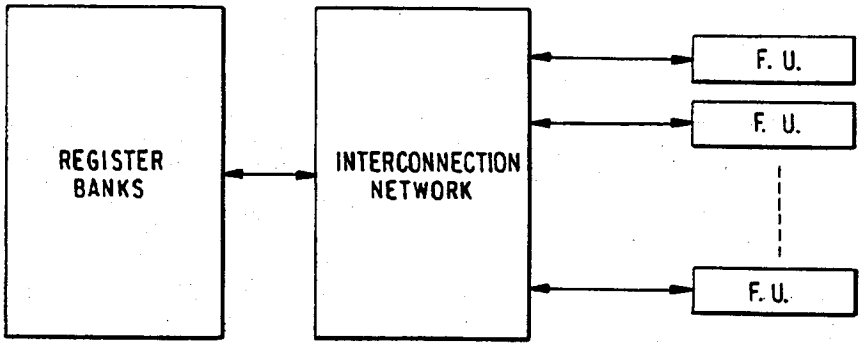



FIG.1B PRIOR ART



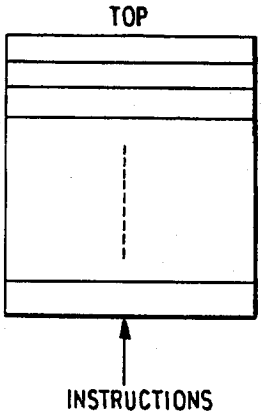




FIG.2A
DISPATCH STACK

FIG.2B
DISPATCH STACK

| TOP | | | | |
|---|---|---|---|---|
| I0, | AD, | F0, | F1, | F0 |
| I1, | AD, | F2, | F3, | F2 |
| I2, | AD, | F0, | F2, | F0 |
| I3, | AD, | F4, | F5, | F4 |
| I4, | AD, | F6, | F7, | F6 |
| I5, | AD, | F4, | F6, | F4 |
| I6, | AD, | F0, | F4, | F0 |

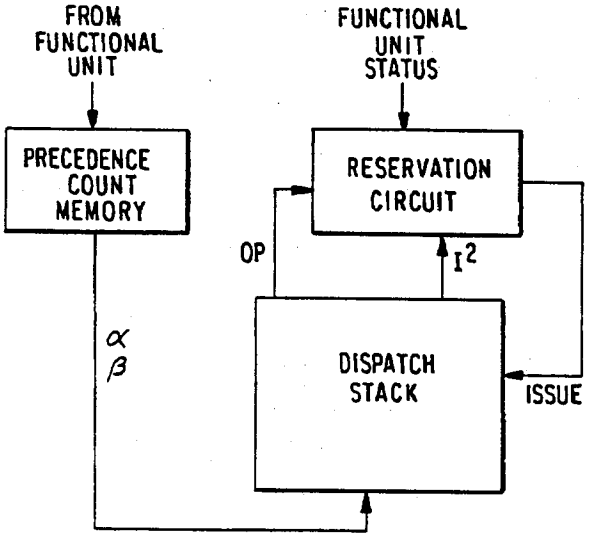



FIG.3

# INSTRUCTION ISSUING MECHANISM FOR PROCESSORS WITH MULTIPLE FUNCTIONAL UNITS

This is a continuation of Ser. No. 539,854, filed on Oct. 7, 1983, now abandoned.

## BACKGROUND OF THE INVENTION

This invention relates to computer architecture and specifically to an instruction issuing mechanism capable of detection of concurrencies in an instruction stream and issuing multiple instructions within a given machine cycle.

The emergence of VLSI technology has stimulated research into the use of execution structures employed by processors having multiple functional units. Such high performance processors are generally partitioned to two sections, an instruction unit (IU) and an execution unit (EU) such is illustrated in FIG. 1A. The IU and EU communicate with each other, with the IU fetching instructions from a memory and formulating and decoding those instructions. The IU is also employed to fetch operands if necessary. Additionally, the IU sends arithmetic/logic commands, that is, the decoded instructions together with a requisite operand to the EU. This invention relates specifically to an instruction issuing mechanism enhancing the throughput, that is, the number of instructions executed per unit of time of the EU.

Within the prior art, processors employing multiple functional units have been designed and implemented. Typical are the CRAY-1 and the IBM 360/91. Reference is made to, R.M. Russell, "The CRAY-1 Computer System" *A.C.M. Communications*, 21: 1, January, 1978, pp. 63–72, and to Srini et al, "Analysis of the CRAY-1S Architecture", *A.C.M.*, 10th *Symposium on Computer Architecture*, June, 1983, pp. 194–206. Reference is also made to R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", *IBM Journal*, 11: 1, January, 1967, pp. 25–33, for description of the IBM 360/91 system.

In accordance with such EU structure employing multiple functional units as shown in FIG. 1B, a bank of registers is installed in the EU to act as a bridge between the fast functional units (F.U.) and a slow main memory. The functional units in the EU perform arithmetic/logic operations on various data types, it being noted that the units are not necessarily identical. For example, reservations stations and a common data bus can also be incorporated. (R. M. Tomasulo, supra). Thus, some of the functional units may be virtual. The registers supply operands to the functional units and receive results from them while at the same time loading from and writing into the main memory.

The IU loads sequences of instructions into an instruction stack from which instructions are issued to and then executed by the functional units. Within prior art systems employing multiple functional units, at most one instruction is issued from the instruction stack during every machine cycle. As a result, the instructions execution rate of such an EU structure cannot be greater than the inverse of the machine cycle time (generally expressed in seconds). Reference is made to R. M. Keller, "Look-Ahead Processors", *Computing Surveys*, 7: 4, December, 1975, pp. 175–195, and J. W. Bowra, et al, "The Modeling and Design of Multiple Function Units Processors", *IEEE Transactions on Computers*, C25: 3, March, 1976, pp. 2102–2210. Specific reference is also made to Srini et al, supra, which indicates that the CRAY-1S system while well balanced suffers from a major drawback. The authors specifically note that the instruction issuing mechanism is a major bottleneck in the CRAY-1S architecture.



## SUMMARY OF THE INVENTION

Given the deficiencies in prior art computer systems employing multiple functional units, it is an object of the present invention to define an instruction issuing mechanism which is capable of detecting concurrencies in an instruction stream and issuing multiple instructions within a given machine cycle and which may be extended to modify the instruction stream.

It is a further object of the invention to define an arithmetic engine implemented in a VLSI environment that substantially enhances the throughput of such a processor.

Yet another object of this invention is to formulate and define an instruction issuing mechanism for arithmetic engines utilizing multiple functional units to achieve high instruction execution rates.

A further object of this invention is to define a dispatch stack component of the instruction issuing mechanism operating in a FIFO mode and detecting instructions that can be issued at each machine cycle.

Still another object of this invention is to define a precedent count memory component of the instruction issuing mechanism to assign alpha- and beta-values for each instruction being loaded into the dispatch stack and to assign general purpose registers to operands to enhance possible execution concurrencies.

These and other objects of the present invention are achieved by an instruction issuing mechanism which detects concurrencies and issues multiple instructions within a given machine cycle. This invention will be described in greater detail by referring the attached drawings and the description of the preferred embodiment that follows.

## BRIEF DESCRIPTION OF THE DRAWINGS

FIG. 1A is a schematic drawing of a black diagram of a processor partitioned into an instruction unit and an execution unit.

FIG. 1B is a schematic drawing of a block diagram of an arithmetic structure utilizing multiple functional units, illustrating the data flow section;

FIG. 2A is a schematic diagram showing the dispatch stack;

FIG. 2B is a schematic diagram illustrating the sequence of instructions deposited into the dispatch stack; and

FIG. 3 is a block diagram of an instruction issuing mechanism in accordance with the present invention.

## DESCRIPTION OF THE PREFERRED EMBODIMENT

Referring to FIGS. 1–3, the instruction format utilized by the present invention will be considered first. As noted herein, the processor is partitioned into two units, the instruction unit (IU) and the execution unit (EU). The IU fetches, formulates, decodes and then forwards arithmetic/logic instructions to the EU. A known format, employed in the CRAY-1 and other systems is as follows:

OP, S1, S2, D (1)

where,

D denotes the register receiving the result of the arithmetic/logic operation;

S1 specifies the register which provides the first of two operands, or the only operand called for;

S2 specifies the register which yields the second of the two operands required; and

OP denotes the arithmetic/logic operation to be performed. In this invention, we consider store register and load register as arithmetic/logic operations.

As a result of this format,

$$D \leftarrow [S1]OP[S2]. \quad (2)$$

If the registers D and S1 are identical, the instruction takes the same form as instructions for the execution units in the IBM 360/91.

The dispatch stack configuration in accordance with the present invention is shown in FIG. 2A. The IU formulates and forwards sequences of 3-register arithmetic/logic instructions as defined herein to the EU. That is, the instruction format utilizes registers D, S1 and S2. These sequences are deposited into a dispatch stack (DS) shown in FIG. 2A. The dispatch stack operates in a first-in first-out (FIFO) fashion. Each stack cell is implemented with a register.

In accordance with known FIFO processing, instructions in the format specified by (1) are sent by the IU to the bottom cell of the DS. This is indicated by the instruction arrow in FIG. 2A. Instructions are then advanced upward in the stack as instructions in an instruction stream are loaded in the DS from the bottom. As a result, when the DS is full in a steady state, the top cell always contains the instruction in the head of the instruction stream. This can be illustrated as follows:

$$S = a_0 + a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 \quad (3).$$

The IU will then fetch operands $a_0 - - - , a_7$ and deposit them into registers F0, - - - , F7, respectively. Additionally, the IU will deposit the following sequence of instructions into the DS.

| |
|---|
| I0, AD, F0, F1, F0 |
| I1, AD, F2, F3, F2 |
| I2, AD, F0, F2, F0 |
| I3, AD, F4, F5, F4 |
| I4, AD, F6, F7, F6 |
| I5, AD, F4, F6, F4 |
| I6, AD, F0, F4, F0 |

where AD indicates an "ADD" operation. It is recognized that the instruction sequence set forth in (4) is merely representative of a number of evaluations for statement (3). The present invention detects and enables execution concurrencies which are inherent in a given statement.

In accordance with the present invention, the IU deposits the sequence of instructions set forth in (4) into an initially empty dispatch stack. Such is shown in FIG. 2B. An instruction tag field representing the left hand most column is added, it being noted that instruction IO occupies the top cell of the DS with subsequent instructions following. The second column contains the OP field, the third column S1, the fourth column S2 and the extreme right hand most colum D. The DS strives to issue arithmetic/logic instructions to the functional units as fast and as many as possible. Such is inherent to take advantage of the presence of multiple functional units in this arithmetic engine. The issuance of such instructions is therefore stopped when one or more of the following three conditions prevail:

(a) the lack of the requisite functional unit;

(b) the lack of the requisite interconnection paths to transmit operand and/or result; and/or

(c) data dependencies among instructions.

In known instruction issuing mechanisms, once an instruction is stopped for any of the three above cited reasons, the flow of subsequent instructions also stops. Thus, in existing instruction issuing mechanisms, the EU examines only the instruction at the head of the instruction stream. Consequently, at most only one instruction can be issued for each machine cycle. As noted, if the top instruction cannot be issued because of the existence of one of the three conditions as set forth above, the flow of the instruction stream is entirely stopped. This deficiency in the prior art tends to degrade the engine output as a consequence of under utilization of available resources, that is, the functional units and the interconnection paths. This deficiency can be illustrated by referring again to FIG. 2B.

With the reservation stations and a common data bus scheme (CDB) the IBM 360/91 utilizes a floating-point execution unit which will examine instruction IO and dispatch it to one of its three virtual adders. In the next machine cycle, the floating-point execution unit examines and dispatches instruction I1 and in the third machine cycle instruction I2 is examined and dispatched. In accordance with this system, for three machine cycles the multiplier will not receive any instruction. Another cause, due to data dependencies, which produces under utilization of functional units will be presented later. This defect is also true with the CRAY-1 system. This defect results in a wasteful under utilization of the functional units which are available in a contemporary EU structure. The situation becomes exacerabated.in the context of VLSI devices due to the continuing decline of hardware cost. That is, since at most one instruction is issued for each machine cycle the instruction execution rate is locked by much less than the inverse of the machine cycle time.

The present invention departs significantly from such known systems by providing the dispatch stack (DS) with capabilities to allow it to examine and issue one or more instructions for each machine cycle. Thus, in accordance with the present invention, the dispatch stack identifies and issues instructions that can be immediately executed with available functional units. The technique of determining data dependencies will now be discussed in the context of the present invention.

An instruction in an instruction stream can be immediately issued to an available functional unit, real or virtual, if it does not have any data dependencies with those preceding instructions which have not yet been completed. For example, referring to FIG. 2B, instruction I2 is data dependent upon instruction I0. This occurs since one of the source registers of I2, F0 is the destination register of I0. Stated differently, I2 utilizes the result of I0 as an operand and therefore it must wait for the completion of instruction I0.

Consequently, it can be generalized that an instruction is data dependent upon a preceding, uncompleted instruction if one of its source registers is the destination register of the latter.

Referring again to FIG. 2B, it can be shown that instruction I2 is data dependent upon instruction I0 in a second sense. Specifically, the destination register of I2, F0, is one of the source registers of I0. Thus, if instruction I2 is issued and completed before I0, I0 may mistakenly utilize the result of I2 as one of its operands. Consequently, it may also be generalized that an instruction is data dependent upon a preceding, uncompleted instruction if its destination register is a source register of the latter.

These two generalizations allow enrichment of the entries in the dispatch stack. The resultant fields are given below:

$$\text{Instruction tag, OP, S1, } \alpha(S1), \text{ S2, } \alpha(S2), \text{ D, } \beta(D), I^2 \quad (5)$$

where $\alpha(Si)$ represents the number of times that a particular register Si is used as a destination register in preceding, uncompleted instructions;

$\beta(D)$ represents the number of times that register D is designated as a source register in preceding, uncompleted instructions; and

$I^2$ represents the issue index field to be delineated herein.

The sequence of instructions entered into the dispatch stack DS, shown in FIG. 2B, is represented as follows:

CHART I

| Instruction Tag. | OP. | S1. | $\alpha$(S1). | S2. | $\alpha$(S2). | D. | $\beta$(D). | $I^2$ |
|---|---|---|---|---|---|---|---|---|
| I0 | AD | F0 | 0 | F1 | 0 | F0 | 0 | 0 |
| I1 | AD | F2 | 0 | F3 | 0 | F2 | 0 | 0 |
| I2 | AD | F0 | 1 | F2 | 1 | F0 | 1 | 3 |
| I3 | AD | F4 | 0 | F5 | 0 | F4 | 0 | 0 |
| I4 | AD | F6 | 0 | F7 | 0 | F6 | 0 | 0 |
| I5 | AD | F4 | 1 | F6 | 1 | F4 | 1 | 3 |
| I6 | AD | F0 | 2 | F4 | 2 | F0 | 2 | 6 |

As shown in CHART I, I0 is at the top of the stack. There is no preceding, uncompleted instruction. Consequently, $\alpha(F0)=\alpha(F1)=\beta(F0)=0$. Instruction I1 has instruction I0 as a preceding, uncompleted instruction but, neither of the source registers associated with instruction I1, that is, registers F2 and F3 are used as the destination register by I0-$\alpha(F2)=\alpha(F3)=0$. Additionally, the destination register of instruction II, register F2 is not employed as a source register by I0—$\beta(F2)=0$. One of the source registers of I2 is F0 which is used by I0 as the destination register. Thus, it can be established that $\alpha(F0)=1$. The other source register of I2 is register F2 which is the destination register of instruction I1. Consequently, $\alpha(F2)=1$. Other $\alpha$- and $\beta$-values can be similarly delineated from the sequence of instructions set forth herein above.

An execution structure having reservation stations in a common data bus, for example, found in the IBM system 360/91 will issue instruction I0 whose two $\alpha$-fields and the $\beta$-field are 0 indicating that there are no data discrepancies preceding uncompleted instructions. Instruction I0 can therefore be immediately executed. The same is also true relative to instruction I1. Instruction I2 is then issued but any of the following conditions will suffice to prevent it from being immediately executed;

(a) $\alpha(F0)=1$ where F0 is used as a destination register by a preceding uncompleted instruction, that is an operand and not yet ready;

(b) $\alpha(F2)=1$, same as above; and

(c) $\beta(F0)=1$, where F0 is used as a source register by a preceding uncompleted instruction—the deposit of the result of instruction I2 may erase an operand needed by a preceding instruction.

Instruction I2 is assigned a reservation station corresponding to a virtual functional unit. Thus, this unit is wasted. To rectify this defect in accordance with the present invention, the Issue Index ($I^2$) for an instruction is as follows:

$$I^2=\alpha(S1)+\alpha(S2)+\beta(D) \quad (6)$$

In order to issue instructions and make the instruction resources used more efficiently, the dispatch stack is scanned from top to bottom. When an instruction with an $I^2$ value of 0 is encountered the issuing mechanism reserves an appropriate functional unit if available and then issues the instruction to it. The implementation of this search and issue operation is in the form of a reservation circuit as shown functionally in FIG. 3.

Considering again CHART I representing the sequence of instructions with $\alpha$, $\beta$, and $I^2$ fields when loaded into the DS. Assume now that there are four functional units which are capable of performing the "ADD" operation and these units are initially free. The search and issue mechanism identifies and issues instructions I0, I1, I3 and I4 concurrently to the four free functional units. This follows the policy rationale set forth herein. It is noted that if the common data bus scheme of the prior art is employed, instructions I0, I1, I2, I3 would have to be issued to the four free functional units. Instruction I2 due to data dependencies indicated in the chart occupies a functional unit without actually being computed. Thus, the unit could be advantageously employed to actually compute I4.

At the completion of an issued instruction, its destination register F0 is used as a "key" to content address the (S1) and the (S2) fields of those instructions which follow it in the dispatch stack and to decrement the appropriate $\alpha$-values by 1.

Similarly its source registers are in turn used to content address the (D) fields of all subsequent instructions and decrement their values.

Illustrating the dispatch stack update process, is CHART II which follows showing that at the completion of the instruction I0 its destination register F0 is used to "content address" the S1 and S2 fields of all the instructions which follow I0 in the DS. The S1 fields of I2 and I6 match the F0 key and their corresponding $\alpha$(S1) fields are decremented by 1. At the same time, the source registers of I0, that are F0 and F1, are used to content address the D fields of all instructions which follow I0 in the DS. The D field of I2 and I6 match the F0 key and their corresponding $\beta$(D) fields are decremented by 1. Moreover, instruction I0 is removed from the DS and subsequent instructions are advanced, that is moved up. Those subsequent instructions and the instruction stream should be brought into occupy empty spaces at the bottom of DS thereby operating in a FIFO mode. Thus, after decrementation and shifting the following chart exists.

CHART II

| Instruction Tag. | OP. | S1. | $\alpha$(S1). | S2. | $\alpha$(S2). | D. | $\beta$(D). | I |
|---|---|---|---|---|---|---|---|---|
| I1 | AD | F2 | 0 | F3 | 0 | F2 | 0 | 0 |

-continued

CHART II

| Instruction Tag, | OP, | S1, | α(S1), | S2, | α(S2), | D, | β(D), | I |
|---|---|---|---|---|---|---|---|---|
| I2 | AD | F0 | 0 | F2 | 1 | F0 | 0 | 1 |
| I3 | AD | F4 | 0 | F5 | 0 | F4 | 0 | 0 |
| I4 | AD | F6 | 0 | F7 | 0 | F6 | 0 | 0 |
| I5 | AD | F4 | 1 | F6 | 1 | F4 | 1 | 3 |
| I6 | AD | F0 | 1 | F4 | 2 | F0 | 1 | 4 |

It will be recognized that similar and in some cases concurrent completions of I1, I3 and I4 will reduce the contents of the DS to that shown as follows:

CHART III

| Instruction Tag, | OP, | S1, | α(S1), | S2, | α(S2), | D, | β(D), | I |
|---|---|---|---|---|---|---|---|---|
| I2 | AD | F0 | 0 | F2 | 0 | F0 | 0 | 0 |
| I5 | AD | F4 | 0 | F6 | 0 | F4 | 0 | 0 |
| I6 | AD | F0 | 1 | F4 | 1 | F0 | 1 | 3 |

Empty spaces ready for subsequent instructions.

The contents of the DS after decrementations and shifts initiated by the completion of Instructions I0, I1, I3 and I4 is therefore illustrated by CHART III.

Instructions I2 and I5 can now be issued and their completion will reduce the $I^2$ value of I6 to 0.

Operating under the assumption that (1) the operands will be available at their designated registers and (2) adequate data paths are available to transmit operands and results, the issue and execution schedule of the sequence of instructions of FIG. 2B will then be:

First-I0, I1, I3, I4

Second-I2, I5

Third-I6

This schedule which produces the shortest computation time would not be detected and followed if a prior art common data bus scheme is employed.

The identification of data dependencies among instructions utilized in the example shown relative to FIG. 2B excludes the following case:

$I_a$: $OP_a$, $S1_a$, $S2_a$, $D_a$

.

.

.

$OP_b$, $S1_b$, $S2_b$, $D_l$

.

.

.

$I_b$: $OP_b$, $S1_b$, $S2_b$, $D_b$

where

$D_a = D_b$ and $D_a \neq S1_b$, $D_a \neq S2_l$ for all i.

The instruction $I_a$ as shown is obviously superfluous since its result is not utilized or needed in subsequent instructions. This case should therefore be excluded from the system compiler software. Should this not be feasible then erroneous consequences will arise if instruction $I_b$ is completed before instruction $I_a$. Nevertheless, this can be prevented by defining β(D) as the number of times that register D is designated as a source register and/or destination registers in preceding uncompleted instructions.

At the completion of an issued instruction, its destination register is used as a "key" to content address the S1, the S2 and the D fields of those instructions which follow it in the dispatch stack. The destination register is also used to decrement the appropriate α- and β-values by 1. Similarly, its source registers are used to content address the D fields of all subsequent instructions and decrement their β-values. Thus, following this methodology instruction $I_b$ will not be issued until the completion of $I_a$.

The precedence count memory (PCM) as shown in FIG. 3 will now be discussed. It has been set forth herein that the IU formulates sequences of 3-register instructions and loads them into the DS. This requires, for example, the assignment of appropriate registers to operands and results. It also requires the determination of α(S1), α(S2), and β(D) for each instruction formulated. These two tasks can be facilitated with the introduction of the precedence count memory (PCM) shown in FIG. 3. The PCM is implemented with a rank of registers, each register corresponding to a general purpose register in the execution unit. Each register has an entry in the PCM. The α-field indicates the number of times that a specific register has been used as a destination register by instructions already in the DS. The β-field denotes the number of times that a specific register has been used as a source register. For example, after seven instructions are loaded into the dispatch stack as shown in Chart I, the PCM will have the entries as depicted below in Chart IV.

CHART IV

| Register | α | β |
|---|---|---|
| F0 | 3 | 3 |
| F1 | 0 | 1 |
| F2 | 1 | 2 |
| F3 | 0 | 1 |
| F4 | 2 | 3 |
| F5 | 0 | 1 |
| F6 | 1 | 2 |

Chart IV is therefore a "snapshot" of the PCM immediately after the DS has been loaded in the manner identified in Chart I.

When an instruction is removed from the DS upon completion, the α-value of its destination register and the β-values of its source registers are each decremented by 1.

When a register is assigned to an instruction as a source register, its α-value in the PCM is used as α(S1) or a α(S2) and its β-value is incremented by 1. When a register is appropriated to an instruction as its destination register, its present β-value is used as the β(D) field and its α-value is incremented by 1.

Thus, in accordance with the present invention a unique instruction issuing mechanism has been defined for execution structures with multiple functional units. This mechanism is capable of detecting concurrencies and then issuing multiple instructions within a given machine cycle. As a result, throughput of such processors is substantially enhanced. While the invention has been defined relative to a preferred embodiment herein, it is apparent that modifications may be practiced without departing from the essential scope of this invention.

I claim:

1. An instruction issuing system for a processor including an execution unit having multiple functional units comprising:

an instruction issuing unit receiving instructions from a memory, operating on instructions and forwarding instructions to said execution unit, said instruc-

tion issuing unit including means for detecting the existence of concurrencies in said instructions received from said memory; and

said instruction issuing unit further including means for issuing multiple instructions and non-sequential instructions to said execution unit within a single processor cycle when a concurrency is detected by said means for detecting the existence of concurrencies in said instructions.

2. The instruction issuing system of claim 1 wherein said means for detecting the existence of concurrencies comprises a dispatch stack receiving instructions from saaid memory and operating in a first-in first-out manner, said dispatch stack receiving instructions having instruction fields of OP, S1, S2, D, where:

OP is the arithmetic/logic operation to be performed,

S1 specifies a register which provides the first of two or the only operand called for,

S2 specifies a register yielding the second operand, and,

D specifies a register receiving the result of the arithmetic/logic operation.

3. The instruction issuing system of claim 2, wherein said means for detecting the existence of concurrencies in said instruction issuing unit further comprises a precedent count memory, said precedent count memory providing fields of a first value ($\alpha$) to instruction fields S1 and S2 indicative of the number of times a register S1(S2) is used as destination register in preceding, uncompleted instructions and, a second value ($\beta$) to register field D indicative of the number of times that register D is designated as a source register in preceding, uncompleted instructions.

4. The instruction issuing system of claim 2 wherein said means for detecting the existence of concurrencies includes a precedent count memory for providing values to each instruction loaded into said dispatch stack indicative of the number of times a register for a particular field is designated as a source register in preceding, uncompleted instructions.

5. The instruction issuing system of claim 3 wherein said means for detecting the existence of concurrencies determines an issue index ($I^2$) for each instruction in said dispatch stack wherein: $I^2=\alpha(S1)+\alpha(S2)+\beta(D)$ such that when an instruction having $I^2=0$ is encountered by said means for detecting the existence of concurrencies, said means for issuing multiple instructions reserves an available functional unit and issues said instruction to it.

6. A method of issuing instructions for a processor having multiple functional units comprising the steps of:

reading in and storing instructions from an instruction stream into a dispatch stack, said instructions having an instruction formate of OP, S1, S2, D, where:

OP is the arithmetic/logic operation to be performed;

S1 is the register which provides the first of two or the only operand called for;

S2 is the register yielding the second operand, and

D is the register receiving the result of the arithmetic/logic operation;

detecting the existence of concurrencies in instructions stored in said dispatch stack and;

issuing multiple instructions and non-sequential instructions within a given processor cycle when the existence of concurrencies is detected.

7. The method of claim 6 wherein said step of detecting further comprises the steps of;

determining the number of times that individual registers in said processor are used as destination registers in preceding, uncompleted instructions, determining the number of times the individual registers in said processor are used as source registers in preceding uncompleted instructions, and providing an indication of the determination in said instruction format for each instruction in said dispatch stack.

8. The method of claim 7 wherein said step of issuing multiple instructions further comprises the step of immediately issuing a first instruction from said dispatch stack to an available functional unit when said instruction does not have any data dependencies with preceding issued instructions which have not yet been completed.

9. The method of claim 8 wherein an instruction is data dependent upon a preceding, uncompleted instruction if one of its source registers is the destination register of the uncompleted instruction.

10. The method of claim 8 wherein an instruction is data dependent upon a preceding, uncompleted instruction if its destination register is a source register of the uncompleted instruction.

11. The method of claim 7 wherein said step of determining comprises providing first values ($\alpha$) to register fields S1(S2) indicative of the number of times register S1(S2) is used as destination registers in preceding, uncompleted instructions and a second value ($\beta$) to register field D indicative of the number of times that register D is designated as a source register in to preceding, uncompleted instructions.

12. The method of claim 11 further comprising the steps of content addressing the S1,S2 and D fields following the completion of an issued instruction and, appropriately decrementing the values of $\alpha$'s, the values of $\beta$'s and updating the dispatch stack by advancing subsequent instructions stored therein and adding new instructions from said instruction stream occupy empty portions at the bottom of said dispatch stack.

13. The method of claim 11 further comprising the steps of updating the values of $\alpha$ and $\beta$ such that when a register is assigned to an instruction as a source register its present $\alpha$ value is used as $\alpha$(S1) or $\alpha$(S2) and its $\beta$ value is incremented by 1 and, when a register is assigned to an instruction as its destination register, its present $\beta$ value is used as the $\beta$(D) field and its $\alpha$-value is incremented by 1 and further when an issued instruction is completed the $\beta$ values of each of its source registers is decremented by 1 and the $\alpha$ value of its destination register is decremented by 1.

14. An instruction issuing system for a processor including an execution unit having multiple functional units comprising:

an instruction issuing unit receiving instructions from a memory, said instruction issuing unit operating on instructions and forwarding instructions to said execution unit, said instruction issuing unit including means for detecting the existence of a plurality of instructions received from said memory which are concurrently executable; and

said instruction issuing unit further including means for issuing multiple instructions and non-sequential instructions to said execution unit within a single processor cycle when concurrently executable instructions are detected by said means for detecting the existence of concurrently executable instructions in said instructions.

15. A method of issuing instructions for a processor having an execution unit with multiple functional units comprising the steps of:

reading in and storing instructions from an instruction stream into a dispatch stack;

detecting the existence of plurality of instructions which are concurrently executable from those instructions stored in said dispatch stack; and

issuing multiple instructions and non-sequential instructions within a given processor cycle when said plurality of concurrently executable instructions are detected.

16. The method of claim 15 wherein said step of detecting further comprises the steps of;

determining the number of times that individual registers in said processor are used as destination registers in preceding, uncompleted instructions, determining the number of times the individual registers in said processor are used as source registers in preceding uncompleted instructions, and providing an indication of the determination in said instruction format for each instruction in said dispatch stack.

17. The method of claim 16 wherein said step of issuing multiple instructions further comprises the step of immediately issuing a first instruction from said dispatch stack to an available functional unit when said instruction does not have any data dependencies with preceding issued instructions which have not yet been completed.

18. The method of claim 15 wherein an instruction is data dependent upon a preceding, uncompleted instruction if one of its source registers is the destination register of the uncompleted instruction.

19. The method of claim 15 wherein an instruction is data dependent upon a preceding, uncompleted instruction if its destination register is a source register of the uncompleted instruction.

* * * * *

# EXHIBIT “2"

*digest of papers*

# COMPCON '95

## Technologies for the Information Superhighway

March 5 – 9, 1995
San Francisco, California



IEEE Computer Society Press
Los Alamitos, California

Washington • Brussels • Tokyo

# Advanced Performance Features of the 64-bit PA-8000

Doug Hunt

Hewlett-Packard Company
Engineering Systems Lab
3404 East Harmony Road, MS#55
Fort Collins, Colorado 80525

*Abstract: The PA-8000 is Hewlett-Packard's first CPU to implement the new 64-bit PA2.0 architecture. It combines a high clock frequency with a number of advanced microarchitectural features to deliver industry-leading performance on commercial and technical applications while maintaining full compatibility with all previous PA-RISC binaries. Among these advanced features are a fifty-six entry instruction reorder buffer to support out-of-order execution, a branch target address cache, branch history table, support for multiple outstanding cache misses and dual integer, load/store, floating point multiply/accumulate, and divide/square root units which allow execution of four instructions per cycle. Together, these features will enable the PA-8000 to sustain superscalar operation on a wide variety of workloads.*

## 1. Introduction

Hewlett-Packard's PA-8000 CPU is designed to deliver industry-leading performance on today's commercial and technical applications while providing a growth path to future 64-bit applications. Maintaining industry-leading performance requires improvement in both clock frequency and the average number of clock cycles per instruction (CPI). Since RISC processors are already capable of starting one operation per cycle, reducing CPI further requires starting more than one operation per cycle. The PA-7100[1], PA-7100LC[2], and PA-7200[3] have already achieved success as two-way superscalar implementations, but adding still more functional units is not useful if the rest of the processor is not capable of supplying those functional units with a continuous stream of operations to perform. With the PA-8000, Hewlett-Packard introduces an entirely new microarchitecture with a carefully chosen set of features designed to *sustain* superscalar operation on real-world applications.

First, the PA-8000 includes two integer ALUs, two shift/merge units, two floating point multiply/accumulate units, two divide/square root units and two load/store units. These functional units are arranged to allow up to four instructions per cycle to begin execution. To supply these functional units with enough work to keep them busy, the PA-8000 incorporates a fifty-six entry Instruction Reorder Buffer (IRB) and a dual ported data cache. In order to keep the buffer as full as possible with instructions to choose from, the instruction fetch unit is designed to supply four instructions per cycle from a single-level, off-chip cache. Finally, in order to maximize the usefulness of the instruction and data caches, a high performance bus interface capable of supporting multiple outstanding cache misses is provided. This set of features will enable the PA-8000 to deliver >360 SPECint92 and >550 SPECfp92 at first release. Figure 1 is a block diagram of the PA-8000 which shows how these components are organized.

### 1.1. Why out of order?

The fundamental difficulty in achieving sustained superscalar operation is finding enough independent work to supply the multiple execution units. One way to handle this problem is to give the burden of finding the parallel work to the compiler by requiring that it order the instructions so that every instruction fetch includes more than one instruction which may be executed at the same time. Unfortunately, there are many situations where the compiler cannot take advantage of potential parallelism because of the limited information available at compile time. The design of the PA-8000 therefore leaves the scheduling of instructions up to the hardware, which can perform more aggressive reordering than the compiler thereby achieving a higher utilization of the functional units.

The PA-7100, PA-7100LC, and PA-7200 gain substantial benefit from their two-way superscalar operation while leaving scheduling to the compiler, but it became apparent during the investigation which led to the development of the PA-8000 that some problems would require a fundamentally different approach as the move was made from two-way superscalar to wider implementations. One of those problems is that legacy code which was compiled with an earlier compiler would not be optimally scheduled and would receive very little benefit from the added superscalar hardware. The more sequential instructions the hardware attempts to execute at once, the more likely it is that the instruction group will contain instructions which depend on one another and cannot be executed simultaneously. The PA-8000 addresses this problem by having the hardware scan a large portion of the program at one time in order to find opportunities for parallel execution, rather than only considering four instructions at once. With its large reorder buffer, the PA-8000 can examine over fifty instructions at one time to find four which are ready to be executed.



1063-6390/95 $4.00 © 1995 IEEE

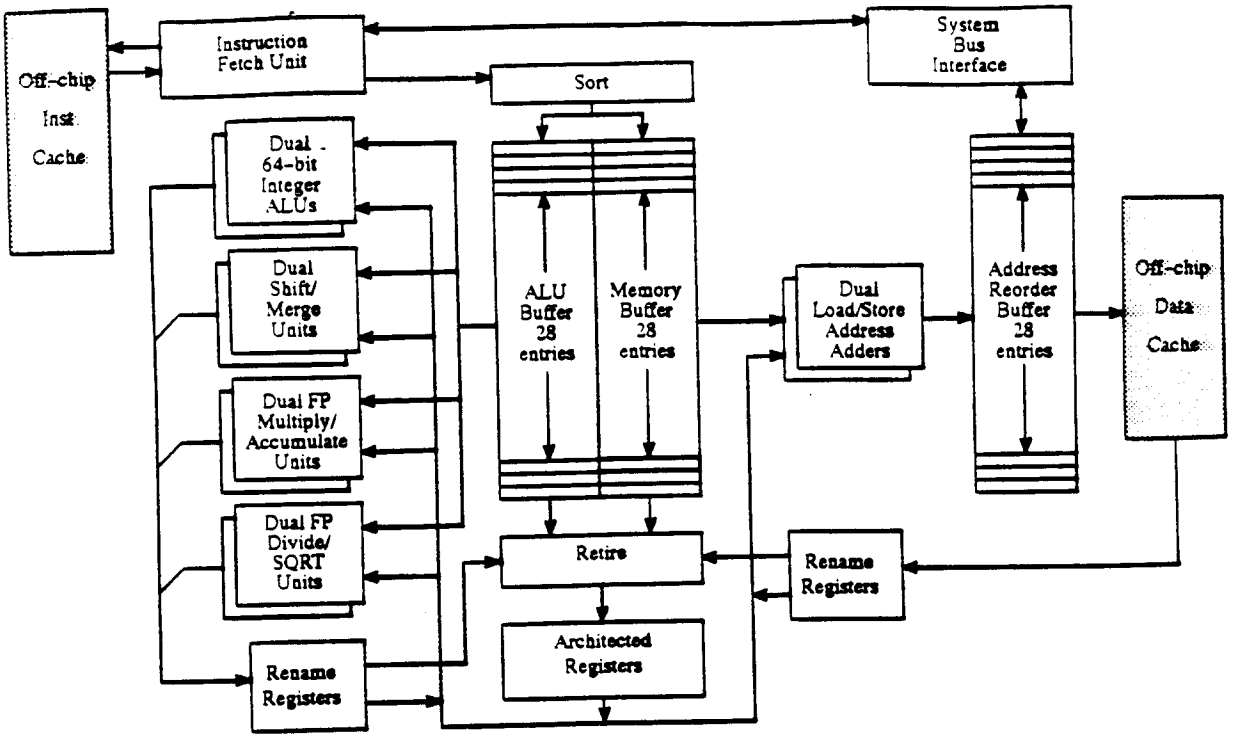



**Figure 1: PA-8000 Block Diagram**

A second problem with leaving scheduling up to the compiler is that the compiler often can only reorder instructions within a narrow window because of such unknowns as flow of control and pointer aliasing. The PA-8000 addresses this problem by performing speculative execution of instructions. Speculative execution is nothing more than guessing what course the program will take and executing instructions from the appropriate path. If it is later discovered that the guess was incorrect, the speculative work is discarded.

The PA-8000 actually performs speculative execution in a number of different ways. First, on every branch fetched, the instruction fetch unit makes an intelligent guess about whether the branch will be taken or not taken and fetches instructions down the appropriate path. When the branch is actually executed, the outcome (either taken or not taken) is compared with the predicted outcome. If the two outcomes do not agree, the correct address (either the branch target address or the inline address) is forwarded to the instruction fetch unit and fetching resumes from that point. All instructions in the IRB younger than the mispredicted branch are discarded.

Another way the PA-8000 performs speculative execution is by executing younger instructions before it is known whether an older instruction will signal an exception (trap). A common example of this situation is the execution of a younger instruction before an older load. The younger instruction must not be allowed to change any program state if the load signals a TLB miss or protection violation. In the PA-8000, the younger instruction is allowed to execute early, but its result is discarded if the older instruction traps.

A third example of speculative execution is the execution of a load before an older store. In this case, it is possible that the load and the store reference the same memory location. This should be a rare event since the compiler tends to keep a value in a register if it will be needed again shortly. However, there are situations where the compiler cannot know that a load and a store point to the same location in memory, especially if the load and store are generated by references through pointers. In the PA-8000, a load may execute before an older store and the hardware checks to see that the load received valid data before the result is committed to the general registers. If a load is determined to have received the incorrect data, the load and all subsequent instructions are flushed from the IRB and refetched.

In each of these three cases, the hardware is able to gain the advantage of performing work early in the common case where the program flow of control proceeds along expected lines and only suffers a performance penalty in the cases where the program flow is not as expected. A compiler often cannot take advantage of the same parallelism because it would have to add so many runtime checks to ensure the reordering was safe that the benefit of reordering the code would be lost.

## 2. Instruction fetch unit

The IRB can only do its job of supplying the execution units with plenty of work if the buffer itself has an adequate

supply of incoming instructions. The PA-8000 instruction fetch unit fetches up to four quadword-aligned instructions per cycle from a single-level off-chip instruction cache. This is the same bandwidth as the maximum execution rate of the functional units. The instruction cache, which is constructed of synchronous SRAMs, has a two cycle latency. Adding the cycle it takes to calculate the target address of a branch means that there is a two cycle penalty for fetching the target of a correctly predicted taken branch from this cache.

To reduce the penalty for taken branches, the PA-8000 incorporates a thirty-two entry fully associative Branch Target Address Cache (BTAC) which associates the address by which a branch is fetched with the address of the target of the branch for branches which are predicted taken. On every instruction fetch, the address sent to the instruction cache is also sent to the BTAC. Whenever the BTAC signals a hit, the address supplied by the BTAC is used as the next fetch address. This means that correctly predicted taken branches which hit the BTAC suffer no penalty, since the quadword containing the target of the branch will arrive on chip the cycle after the branch itself arrives. A new entry is inserted into the BTAC each time a predicted-taken branch is fetched for which there is not already an entry in the BTAC. This insert does not cause any additional instruction fetch penalty. A "round robin" replacement policy is employed.

### 2.1. Branch prediction

To achieve sustained superscalar operation, it is important that the number of mispredicted branches be minimized. To improve branch prediction accuracy, two different methods of branch prediction are provided: static and dynamic. In static prediction mode, the fetch unit follows the following policy: For most conditional branches, backward branches are predicted "taken" and forward branches are predicted "not taken". For the common compare and branch instruction, a hint is specifically encoded in the instruction to tell the instruction fetch unit which way to predict the branch. Compilers using either heuristic methods or Profile Based Optimization (PBO) can rearrange code segments or use the hinted branches to effectively communicate branch probabilities to the hardware.

In dynamic prediction mode, a 256-entry Branch History Table (BHT) is consulted to determine which way each branch should be predicted. Each entry in the BHT is a three-bit shift register which records the last three outcomes (taken or not taken) of a given branch. If a majority of the last three executions were actually taken, the fetch unit predicts that the branch will be taken again. This table is only updated as branch instructions are retired in order to prevent corrupting the history information with speculative executions of the branch.

The branch prediction mode used (either static or dynamic) is controlled on a page-by-page basis by an extra bit in each entry of the TLB. Thus, it is possible for programs compiled with PBO to take advantage of the profile information, while programs which have not been profiled use dynamic prediction. It is also possible for shared libraries to be profiled, if appropriate, in which case even non-profiled applications will gain the benefit of the profiling of the libraries. This also has the advantage that the library code will not displace the history information in the BHT, improving its effectiveness for the main body of the program.

Note that it is possible for the BTAC to signal a hit (indicating a predicted-taken branch) when the BHT signals that the branch should be predicted not-taken or that an older branch in the group should have been taken instead. In this case, the corresponding entry in the BTAC will be deleted to prevent another hit of the BTAC on that branch.

### 2.2. Why no on-chip instruction cache?

An instruction cache can improve performance in two main ways: it can reduce the latency of instruction fetches and it can be designed to provide more bandwidth to the processor than the next level of the memory hierarchy can provide. HP's low-cost processor, the PA-7100LC, is the only HP PA-RISC processor to include an on-chip instruction cache. Since the design of the PA-7100LC was driven by the need to minimize overall system cost, a single combined off-chip cache was provided. This necessitated including an on-chip instruction cache to provide sufficient instruction fetch bandwidth to the execution units without overly impacting the data cache performance.

The PA-8000, on the other hand, is designed to maximize performance. For many real-world applications, especially some commercial applications such as transaction processing, delivering high performance requires a larger instruction cache than can be included on-chip. Furthermore, a four-way superscalar design such as the PA-8000 requires a wide connection to this large instruction cache to avoid substantial fetch penalties. Once the decision has been made to design a high-bandwidth connection to a large off-chip cache, an on-chip cache provides no added benefit from a bandwidth perspective.

As far as latency is concerned, the only time the latency of an instruction fetch matters is when a branch is involved. This is because the instruction fetch unit does not need to see the incoming instructions to calculate the next address to fetch if the program is executing sequentially. As mentioned earlier, the BTAC avoids the taken branch penalty for most of the taken branches which are encountered. This means that the only time the reduced latency provided by an on-chip cache would come into play is in the case of a mispredicted branch. Since the aggressive design of the off-chip cache path resulted in a two-cycle latency, an on-chip cache could only save one cycle in the rare event of a mispredicted branch. This being the case, the die area was used for more effective performance features, particularly the fifty-six entry instruction reorder buffer, rather than for an on-chip cache.

## 3. Instruction reorder buffer

The fifty-six entry Instruction Reorder Buffer (IRB) is physically organized as two separate buffers of twenty-eight entries each. One buffer is used to hold instructions which are destined for either the integer units or the floating point units and the other buffer holds both integer and floating point load and store instructions. Some instructions are inserted into both buffers. These instructions are: (1) load-and-modify instructions, for which the modify is handled

by an integer ALU; (2) branches, which go into both buffers to help in recovery from mispredicted branches; and (3) certain system control instructions.

Insertion of instructions into the two buffers in the IRB is controlled by the sort unit. This unit receives the four instructions from the instruction fetch unit and routes each of them to one or both of the buffers in the IRB. Each buffer can accept up to four instructions per cycle, so an arbitrary collection of four instructions may be inserted simultaneously.

Once an instruction has been inserted into a slot of the IRB, the hardware watches each of the instructions launching to the functional units and checks to see whether any of them supplies any of the operands which the instruction in the slot requires. Once the last instruction upon which the slot is waiting has been launched, the slot begins to arbitrate for launch to the functional units. Even though the instructions are segregated into two different buffers, all of the launch information is visible to both buffers. No extra penalty is incurred for bypassing information from instructions in one buffer to instructions in the other buffer.

Up to two instructions per cycle may be launched from each buffer in the IRB. Arbitration in each buffer is handled in two groups. All of the even-numbered slots in the ALU buffer which are ready to launch arbitrate for launch to alu0 and all of the odd slots arbitrate for launch to alu1, and similarly for the memory buffer. In each buffer, the even-numbered slot containing the oldest instruction and the odd-numbered slot containing the oldest instruction win arbitration and are launched to the execution units or the address adders.

### 3.1. Retirement

Instructions are removed from the IRB in program order after they have successfully executed or their trap status is known. Up to four instructions may be retired per cycle. At retire time, the contents of the rename register associated with a given instruction are committed to the architected registers, and store data is forwarded to the store queue (discussed later). If an instruction needs to signal a trap, the trap parameters are recorded in the architected state and the appropriate trap vector is forwarded to the instruction fetch unit which begins fetching from that address. The fact that instructions are retired in program order and that traps are signalled when an instruction retires enables the PA-8000 to provide a precise exception signalling model.

## 4. Loads and stores

A frequent cause of pipeline stalls in pipelined in-order machines is that instructions must often wait for the result of preceding load operations. Previous implementations of PA-RISC[4] have implemented stall-on-use and hit-under-miss policies to avoid these penalties in the case of data cache misses. Unfortunately, these techniques are insufficient to avoid large performance penalties when more instructions are executed simultaneously. In fact, load/use penalties can be a serious performance limiter even when loads hit the data cache. As applications demand larger caches to support bigger working sets and as the operating frequency of processors increases, the number of clock cycles required to load data from the data cache increases. This problem is exacerbated in a wide superscalar machine because the distance which must separate a load from the use of its data to avoid a stall is likely to be more than the compiler can accommodate (refer to section 1.1).

Out-of-order execution is obviously a substantial advantage in being able to avoid load/use penalties. Given that the PA-8000 can dynamically schedule instructions over a window of more than fifty instructions, the hardware can look beyond the instructions dependent on a load and find other instructions ready to be executed. This is especially helpful in the case of data cache misses since, if the hardware finds another load or store which misses the data cache, that miss will also be issued on the system bus. Since the two miss transactions are overlapped, the total performance penalty is less than the cost of two sequential data cache misses. The PA-8000 can support up to ten such outstanding data cache misses at one time. This is accomplished without sacrificing a strongly ordered programming model.

When a slot containing a load or store operation determines that the operands required for calculating its effective address are available, it arbitrates for launch to the address adders, just as instructions in the ALU buffer launch to the integer and floating point units. Once the address is calculated, the address is stored in the address reorder buffer (ARB). The effective address is also sent to the TLB, which is dual ported, and the physical page number associated with the effective address is also stored in the ARB. The ARB is twenty-eight slots deep, and each slot of the ARB is associated with a slot of the memory buffer in the IRB.

The ARB is the interface to the dual-ported, single-level off-chip data cache. The two ports of the data cache are connected to separate banks of synchronous SRAMs, one of which contains even-numbered doublewords and the other odd-numbered doublewords. The data cache may be up to four Mbytes in size.

Once an address has been sent to the ARB, if no other instruction is arbitrating for access to the appropriate bank of the data cache, the cache access is immediately launched to the RAMs. In this case, load data arrives back on the chip in time for a dependent instruction to launch on the third cycle after the load launched to calculate its effective address.

In the event that a load cannot immediately access the data cache port it needs, it begins to arbitrate for access on each successive cycle until it wins arbitration. Arbitration is granted based on the age of the originating instruction, not the length of time a load has been in the address reorder buffer. Instructions in the IRB are informed of the status of loads in progress so that instructions waiting for load data do not arbitrate for launch until the load has won access to the data cache. In this way, the execution units continue to work on other, younger instructions which do have all their operands available.

Store instructions merely perform a tag lookup at the time when a load would read the cache. In the event that the store misses the cache, it proceeds to issue its miss to the system bus. Store data is copied from the register file to the store queue at retire time.

The store queue is a structure which can hold up to eleven doublewords of write data for each bank of the data cache. The store queue uses idle cycles, or cycles when oth-

er stores are performing tag lookups, to perform its writes to the data cache. By deferring cache writes to otherwise idle cycles, loads are less likely to be held off from accessing the cache due to contention. Another benefit of the store queue is that stores of less than doubleword size may be merged into a single cache write, thus improving cache utilization. Loads may bypass data directly from the store queue.

Store-to-load dependency checking is implemented through address comparisons performed in the ARB. When a store instruction calculates its effective address, all younger load instructions which have completed their access to the cache compare their address against the store address. If the load detects a match, the load and all younger instructions are flushed from the IRB and re-executed. When a load calculates its effective address, all older stores compare their address against the load address and, if they detect a match, the load waits until the store data is available.

Loads and stores to the I/O address space and semaphore instructions do not issue transactions on the system bus until they are the oldest instructions in the IRB so that they do not issue speculatively.

Support for explicit data cache prefetching is implemented in the PA-8000 via loading to general register zero. This operation may cause a data cache miss which will be issued on the system bus, but the instruction will not cause a trap if the access misses the TLB or if the access fails protection checks. (In these cases, the instruction executes as a NOP.) Unlike ordinary loads, a load to general register zero may retire before a data cache miss it has initiated has been returned from memory. The return data will still be written into the data cache. If a subsequent ordinary load is encountered before the data is returned from memory, the ordinary load is informed that the miss has already been issued, and a second miss to the same address is suppressed.

### 4.1. TLB

The ninety-six entry Translation Lookaside Buffer (TLB) of the PA-8000 is fully dual-ported in order to support two data cache accesses per cycle without requiring these two accesses to be to the same page. Each entry in the TLB may map any power-of-four sized segment of memory from 4 Kbytes to 16 Mbytes. In addition to the main TLB, the instruction fetch unit maintains a buffer of four translations for its use. Whenever the fetch unit misses its set of translations it sends a request to the main TLB to perform a translation on its behalf. This new translation is then inserted into the fetch unit's buffer. Translations for loads and stores take precedence over translations for instruction fetches. A bypass path is provided so that, in the event a mispredicted branch misses the translation buffer in the instruction fetch unit, the translation from the main TLB is available in time to perform the cache tag compare for the first fetch at the new address.

### 4.2. Multiprocessing support

The PA-8000 supports a snoopy multiprocessor cache coherency protocol. No external logic is required for up to eight-way multiprocessing. Support is also provided for higher-order multiprocessing using a hierarchical bus structure.

Incoming Cache Coherency Checks (CCCs) take just one cycle from one bank of the data cache to perform their snoop of the data cache. This very low cost for CCCs means that, even on a fully saturated system bus, CCCs consume no more than 10% of the available data cache bandwidth if they all miss. Even in a system where the system bus is saturated and every CCC *hits dirty*, the CCCs consume no more than 50% of the victim processor's data cache bandwidth. No other penalty is paid by the victim processor. Instruction fetching is unaffected, the ALUs are unaffected, and the other data cache cycles are fully available.

CCCs use the same address comparison mechanism to implement strong ordering between processors as is used to detect store-to-load dependencies within a single processor. If an incoming CCC matches a load or store in the ARB, that load or store is flushed and re-executed.

## 5. Execution units

The PA-8000 integer units implement the new 64-bit functionality in PA2.0 while maintaining full compatibility with existing 32-bit binaries. The new 64-bit operations may be executed even by a program executing with 32-bit addressing, so compilers can take advantage of the wider datapath to improve performance even on 32-bit code. Each integer unit includes shift/merge logic so that it can execute any of the extract or deposit instructions as well as the normal arithmetic operations. A branch adder is also associated with each integer unit.

The floating point hardware in the PA-8000 consists of two multiply-and-accumulate (MAC) units and two divide/square root units. The MAC units perform the very common operation D = A*B+C. These units have a latency of three cycles and are fully pipelined so they may accept a new operation every cycle, giving a maximum throughput of four FLOPs per cycle. Multiply operations and add operations are also handled by the MACs.

The divide/square root units have latencies of 17 for single-precision operations and 31 for double-precision operations. These units are not pipelined, but other FLOPs may execute on the MACs while the divide/square root units are busy. The combination of multiple execution units, a dual-ported data cache, support for up to ten pending data cache misses, and explicit data prefetching support provides exceptional floating point performance, even on workloads whose working sets are larger than the data cache.

## 6. System interface (Runway)

The PA-8000 interfaces directly to the Runway bus, which is the same bus used by the PA-7200[3]. This is a 64-bit multiplexed address/data split transaction bus. The bus supports the full 40-bit physical address space of the PA-8000, allowing access to as much as 960 gigabytes of RAM. Arbitration takes place on separate wires, so it does not consume any bus cycles.

The bus interface logic of the PA-8000 allows up to ten data cache misses, one instruction cache miss, and one instruction cache prefetch to be pending at the same time for the local processor. These transactions may return in any order, allowing improved memory system performance

in the presence of bank contention. Instruction cache prefetches are initiated by the bus interface itself by fetching the next sequential line whenever a cache miss is received from the instruction fetch unit.

The bus interface supports cpu:bus frequency ratios of 1:1, 4:3, 3:2, 5:3, 2:1, 7:3, 5:2, and 3:1. That is, the processor uses a clock which is at least as fast as the system bus clock and may be up to three times as fast as the system bus. This flexibility allows the design of a wide range of products combining various processor speeds and bus frequencies to produce the highest possible system performance from the available subsystems.

## 7. Other performance features

In addition to those features already outlined, the PA-8000 implements a number of new features added to PA-RISC in PA2.0 to improve performance:

- A 22-bit displacement instruction address relative branch to reduce the cost of procedure calls to distant procedures
- A short pointer external branch to reduce the overhead of branching between spaces
- A fast TLB insert mechanism to reduce the cost of TLB misses
- Longer (16-bit) displacement load and store operations
- New variants of the Floating-Point Compare and Floating-Point Test instructions to allow multiple independent conditions to be tested

### 7.1. Performance monitoring

Perhaps one of the most important performance features of the PA-8000 is the hardware that has been included for performance monitoring and debug support. Hardware has been included that can match specified patterns of instruction opcode, address, cache hit/miss status, and branch mispredictions. These match signals can be combined using an on-chip state machine to detect specific sequences of events. Finally, the state machine can cause one of four event counters to increment. The event counters may also be programmed to increment based on a number of other control signals that indicate what the processor is doing at any given time. This hardware will be used by HP's performance analysis group to identify opportunities for compiler improvements to achieve even higher performance with the PA-8000. It will also be used to evaluate additional features which could be of benefit in future processor designs

## 8. Conclusions

Hewlett-Packard's PA-8000 CPU is designed to deliver industry-leading performance on both commercial and technical applications and provide a growth path for future 64-bit applications. It achieves its high performance through a combination of high clock frequency and sustainable superscalar operation. Sustainable superscalar operation is accomplished by matching dual integer, floating point multiply/accumulate, and divide/square root functional units with a very deep instruction reorder buffer, a high performance instruction fetch unit, dual ported data cache and and support for multiple pending cache misses.

## Acknowledgements

Many people have been involved in the development of the PA-8000 and the author would like to thank all of them for the tremendous effort they have put in to make this processor possible. Special recognition is warranted for those individuals who worked on the project from its earliest days and who set the direction of the design: Gregg LeSartre, Jon Lotz, Don Kipp, Darius Tanksalvala, and Steve Mangelsdorf.

## References

[1] E. DeLano, et al, "A High Speed Superscalar PA-RISC Processor", Compcon Digest of Papers, February 1992, pp. 116-121.

[2] P. Knebel, et al, "HP's PA7100LC: A Low-Cost Superscalar PA-RISC Processor", Compcon Digest of Papers, February 1993 pp. 441-447.

[3] G. Kurpanek, et al, "PA7200: A PA-RISC Processor with Integrated High Performance MP Bus Interface", Compcon Digest of Papers, February 1994, pp. 375-382.

[4] R. B. Lee, "Precision Architecture", IEEE Computer, Vol. 22, No. 1, January 1989, pp. 78-91.

# EXHIBIT “3"

## FP 13.2: A 56-Entry Instruction Reorder Buffer

Neela Bhakta Gaddis, Joseph R. Butler, Ashok Kumar, William J. Queen

Hewlett-Packard Company, Fort Collins, CO, Cupertino, AC

A speculative execution high-end PA-RISC CPU has two 28-entry out-of-order instruction reorder buffers (IRBs), one for alu/floating point operations and one for memory operations [1, 2]. The IRBs are capable of inserting any combination of four instructions per cycle. Each cycle, the IRBs launch up to four instructions for execution, two from the ALU IRB and two from the MEM IRB. Up to four instructions (two from each IRB) retire each cycle. The insert, launch and retire mechanisms of this out-of-order machine contain 850k transistors in 52.6mm².

Dependency clearing and launch arbitration are complications inherent to an out-of-order machine. This processor manages over a dozen different types of dependencies for as many as 56 instructions. Techniques used to accomplish dependency clearing and execution arbitration in a single state are described below.

Operand dependencies exist when the source data of one instruction is the result of an earlier instruction. Because of their high-frequency, operand dependencies are tracked using a broadcast mechanism for maximum performance. A register scoreboard is used to determine dependencies when an instruction is inserted into an IRB entry. This dependency setting requires 5b of comparison per inserting instruction (4) per operand (3 )for each entry of the IRBs (56) for a total of 3360 comparators. Dependency clearing at launch requires another 3360 comparators. Clearly, the challenge for operand dependency tracking is the design of a fast, dense comparator. Figure 1 shows the comparator used for dependency clearing. The launching instructions drive their tags on the dual-rail lines inH and inL. A mismatch will drive the wire-ORed CMP line. In each entry there is one CMP line for each of the four launch buses. If none of the CMP lines is driven, the dependency is cleared. The ALU and MEM IRBs utilize 1mm² for these 3360 comparators.

Carry-borrow (CB) dependencies exist when an instruction depends on the CB bits of the PSW. Due to less frequent occurrence, CB dependencies are tracked using a propagate system. This approach trades off increased latencies for area savings. CB dependencies are identified at insert. The most recently inserted entry indicates to the four inserting entries the presence of a CB writing instruction in the IRB. Any inserting instruction can change the CB status for the following instructions (Figure 2). Clearing of CB dependencies starts with the launch of an instruction writing these bits. The launching entry drives its rename tag to the instructions immediately following. The tag propagates through the IRB at a rate of up to two entries per cycle. Once the tag reaches an entry, its dependency is cleared. If an instruction writes CB data, it stops the tag propagation, and drives its own tag upon launching. This system tracks 28 dependencies in 0.7mm².

In an out-of-order processor, stores and loads to the same cache index must recognize each other's presence. This processor large dual-ported data cache requires that two-cache indices be compared to the index of each reference active in the 28-entry address reorder buffer (ARB), where each comparison is effectively 21b wide. Area constraints require compactness, while performance requires speed. The solution is the comparator bit slice circuit in Figure 3. This circuit has the advantages of low input-capacitance, high-density, few wires and low-latency. This comparator design scales well both in width and total number of comparators. This circuit selects the appropriate cache-port index bus (INDEX0, INDEX1) and performs a bit compare during the first phase. On the following phase, the bit comparisons are wired ORed to indicate a match or mismatch.

Once an instruction entry has cleared all of its dependencies through the mechanisms discussed above, it requests permission to launch. Design of the launch priority encoder to evaluate which four of the fifty-six entries launches utilizes a small performance trade-off to gain significant reductions in cycle time and area. Of the four launching instructions, two come from ALU IRB and two from MEM IRB. Each IRB is also divided into two halves. The instructions from the even half execute on the even units, odd, on odd units. The 28 instruction entries in each IRB are further divided into 3 banks of eight instructions each (four odd, four even) and 1 bank of 4 instructions (two odd, two even) (Figure 4). The first entry ready to launch in the oldest requesting bank wins the arbitration for launch in each half of the IRB.

To gain speed, intrabank and interbank grants are evaluated concurrently. Each bank evaluates which entry wins the arbitration within that bank using dual rail dynamic logic. The circuit in Figure 5 is essential in converting the single-ended dynamic request signal to a pair of complementary dynamic signals. Interbank grant calculations begin with each bank dynamically indicating to the other banks if it has an instruction requesting to launch. Each bank then incorporates its relative age with request information from the other banks to determine if it will grant a bank launch. In the last phase of evaluation, the intrabank and interbank grants are combined to determine which instruction in each half of each IRB will launch. Figure 6 is a micrograph.

This RISC CPU utilizes different circuit techniques and dependency tracking mechanisms in the instruction reorder buffer to achieve speed, area, performance and time-to-market goals.

*Acknowledgments:*

The authors acknowledge physical design contributions by: B. Arnold, P. Bodenstab, S.Chapin, W. Jaffe, W. Kever, Y. Kim, K. Koch, T. Lelm, J. Lotz, R. Mason, S. Naffziger, M. Storey, and T. Xu.

*References:*

[1] Hunt, D., "Advanced Performance Features of the 64-bit PA-8000" Digest of Papers, Compcon 1995

[2] Lotz, J., et al., "A Quad-Issue Out-of-Order RISC CPU" ISSCC Digest of Technical Papers, pp. 212-213, Feb., 1996.

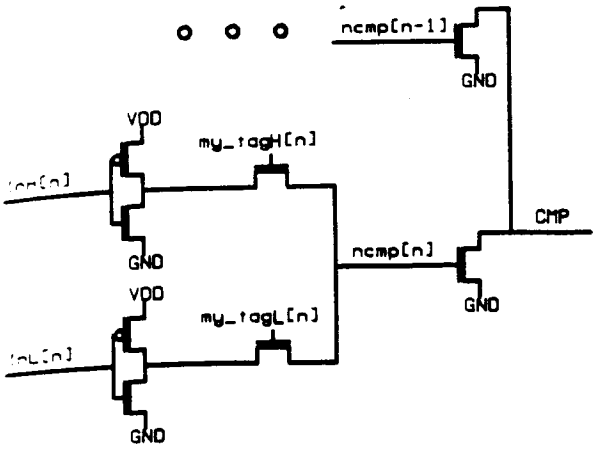



Figure 1: Basic operand comparator cell.

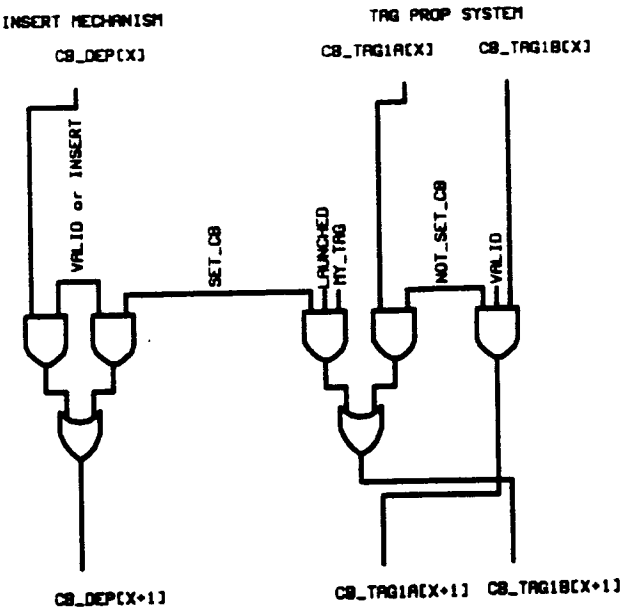



Figure 2: Bit slice for CB dependencies.

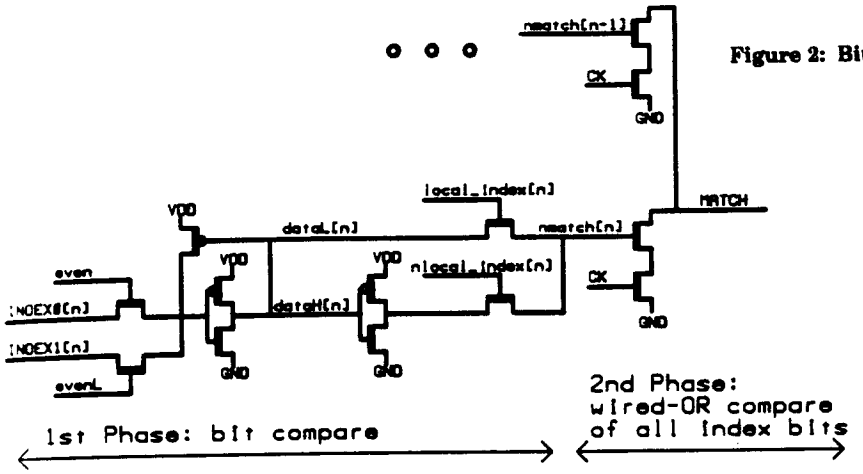



Figure 3: Bit slice used in memory dependency compares.

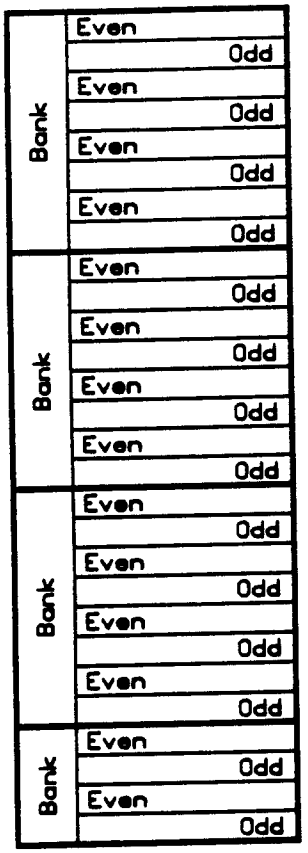



Figure 4: Organization of ALU/MEM IRB.

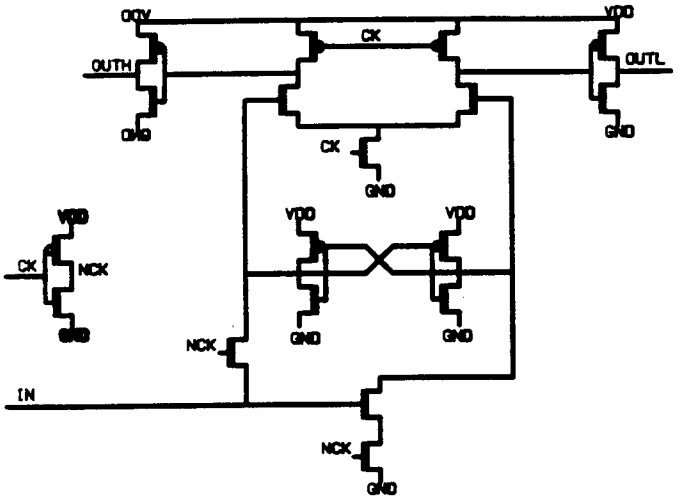



Figure 5: Dual-rail dynamic converter.
Figure 6: See page 447.

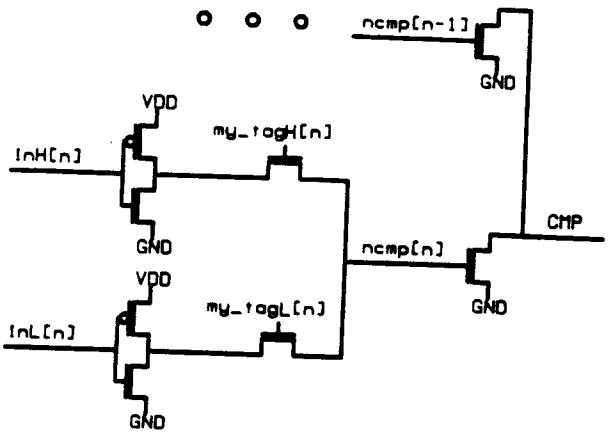



Figure 1: Basic operand comparator cell.

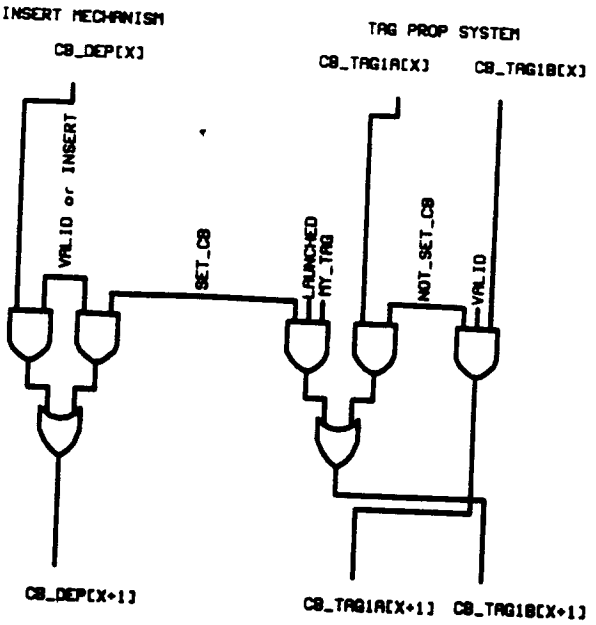



Figure 2: Bit slice for CB dependencies.

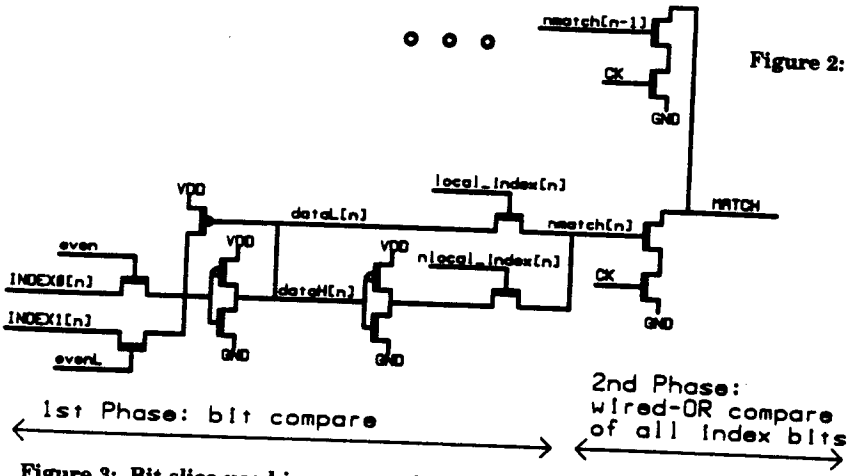



Figure 3: Bit slice used in memory dependency compares.

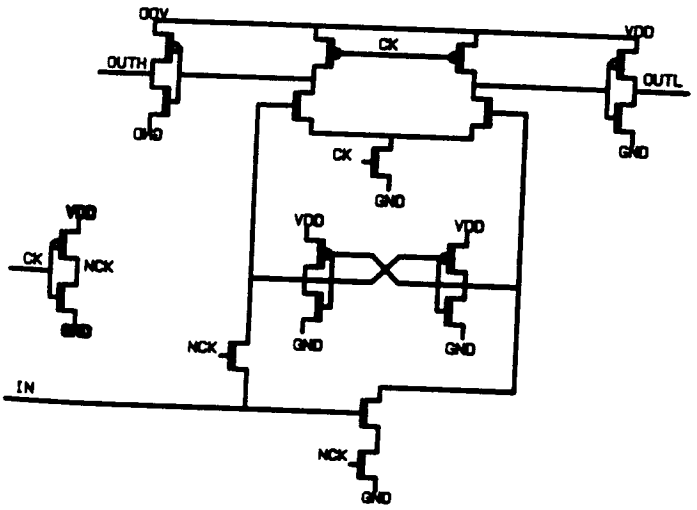



Figure 5: Dual-rail dynamic converter.
Figure 6: See page 447.

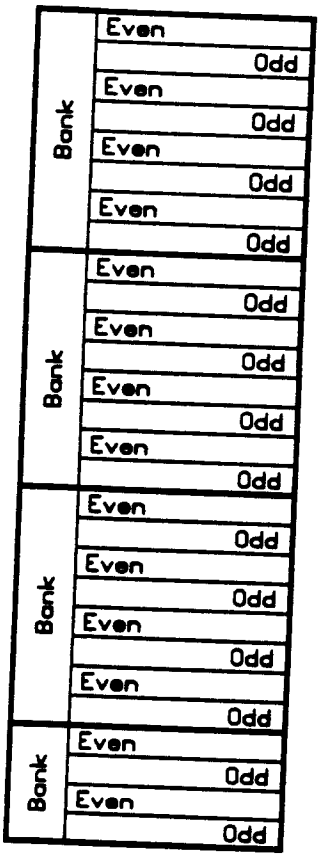



Figure 4: Organization of ALU/MEM IRB.